

This Page Is Inserted by IFW Operations
and is not a part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

**As rescanning documents *will not* correct images,
please do not report the images to the
Image Problem Mailbox.**

Refine Search

Search Results -

Term	Documents
(27 AND 6).USPT.	0
(L6 AND L27).USPT.	0

Database:

US Pre-Grant Publication Full-Text Database
 US Patents Full-Text Database
 US OCR Full-Text Database
 EPO Abstracts Database
 JPO Abstracts Database
 Derwent World Patents Index
 IBM Technical Disclosure Bulletins

Search:

L28

Refine Search

Recall Text

Clear

Interrupt

Search History

DATE: Thursday, May 20, 2004 [Printable Copy](#) [Create Case](#)

Set Name Query
side by side

Hit Count Set Name
result set

DB=USPT; PLUR=YES; OP=ADJ

<u>L28</u>	l6 and L27	0	<u>L28</u>
<u>L27</u>	termination flag	182	<u>L27</u>
<u>L26</u>	l2 and l7 and l8 and l12 and l13	0	<u>L26</u>
<u>L25</u>	l2 and l6 and l7 and l8 and l13	3	<u>L25</u>
<u>L24</u>	l13 and L22	0	<u>L24</u>
<u>L23</u>	l7 and l8 and L22	0	<u>L23</u>
<u>L22</u>	L12.ab.	17	<u>L22</u>
<u>L21</u>	execut\$ and l18	1	<u>L21</u>
<u>L20</u>	l18 and frame\$	0	<u>L20</u>
<u>L19</u>	l18 and exeution frame	0	<u>L19</u>
<u>L18</u>	l17 and block\$	1	<u>L18</u>
<u>L17</u>	l16 and Java and lock\$	1	<u>L17</u>
<u>L16</u>	l8 and L14	1	<u>L16</u>

<u>L15</u>	l7 and L14	0	<u>L15</u>
<u>L14</u>	l12 and L13	1	<u>L14</u>
<u>L13</u>	monitor near2 lock\$	723	<u>L13</u>
<u>L12</u>	terminating near2 thread\$1	455	<u>L12</u>
<u>L11</u>	l7 and l8 and L10	1	<u>L11</u>
<u>L10</u>	l6 and L9	6	<u>L10</u>
<u>L9</u>	termination procedure\$1	740	<u>L9</u>
<u>L8</u>	except\$	921532	<u>L8</u>
<u>L7</u>	command	204689	<u>L7</u>
<u>L6</u>	terminat\$ near3 thread\$1	2071	<u>L6</u>
<u>L5</u>	l3 and L4	8	<u>L5</u>
<u>L4</u>	forcibly	45050	<u>L4</u>
<u>L3</u>	l1 and L2	168	<u>L3</u>
<u>L2</u>	thread\$ and terminat\$	131583	<u>L2</u>
<u>L1</u>	long.in.	6214	<u>L1</u>

END OF SEARCH HISTORY

Refine Search

Search Results -

Term	Documents
(27 AND 6).USPT.	0
(L6 AND L27).USPT.	0

Database:

US Pre-Grant Publication Full-Text Database
 US Patents Full-Text Database
 US OCR Full-Text Database
 EPO Abstracts Database
 JPO Abstracts Database
 Derwent World Patents Index
 IBM Technical Disclosure Bulletins

Search:

L28

Refine Search

Recall Text

Clear

Interrupt

Search History

 DATE: Thursday, May 20, 2004 [Printable Copy](#) [Create Case](#)

 Set Name Query
 side by side

 Hit Count Set Name
 result set

DB=USPT; PLUR=YES; OP=ADJ

<u>L28</u>	l6 and L27	0	<u>L28</u>
<u>L27</u>	termination flag	182	<u>L27</u>
<u>L26</u>	l2 and l7 and l8 and l12 and l13	0	<u>L26</u>
<u>L25</u>	l2 and l6 and l7 and l8 and l13	3	<u>L25</u>
<u>L24</u>	l13 and L22	0	<u>L24</u>
<u>L23</u>	l7 and l8 and L22	0	<u>L23</u>
<u>L22</u>	L12.ab.	17	<u>L22</u>
<u>L21</u>	execut\$ and l18	1	<u>L21</u>
<u>L20</u>	l18 and frame\$	0	<u>L20</u>
<u>L19</u>	l18 and exeution frame	0	<u>L19</u>
<u>L18</u>	l17 and block\$	1	<u>L18</u>
<u>L17</u>	l16 and Java and lock\$	1	<u>L17</u>
<u>L16</u>	l8 and L14	1	<u>L16</u>

<u>L15</u>	l7 and L14	0	<u>L15</u>
<u>L14</u>	l12 and L13	1	<u>L14</u>
<u>L13</u>	monitor near2 lock\$	723	<u>L13</u>
<u>L12</u>	terminating near2 thread\$1	455	<u>L12</u>
<u>L11</u>	l7 and l8 and L10	1	<u>L11</u>
<u>L10</u>	l6 and L9	6	<u>L10</u>
<u>L9</u>	termination procedure\$1	740	<u>L9</u>
<u>L8</u>	except\$	921532	<u>L8</u>
<u>L7</u>	command	204689	<u>L7</u>
<u>L6</u>	terminat\$ near3 thread\$1	2071	<u>L6</u>
<u>L5</u>	l3 and L4	8	<u>L5</u>
<u>L4</u>	forcibly	45050	<u>L4</u>
<u>L3</u>	l1 and L2	168	<u>L3</u>
<u>L2</u>	thread\$ and terminat\$	131583	<u>L2</u>
<u>L1</u>	long.in.	6214	<u>L1</u>

END OF SEARCH HISTORY



[Subscribe \(Full Service\)](#) [Register \(Limited Service, Free\)](#) [Login](#)

Search: ☒ The ACM Digital Library ☐ The Guide

forcibly terminating a thread and JAVA and exception and lock



[Feedback](#) [Report a problem](#) [Satisfaction survey](#)

Terms used

forcibly terminating a thread and JAVA and exception and lock monitor

Found 18,975 of 132,857

Sort results by

relevance



[Save results to a Binder](#)

[Try an Advanced Search](#)

[Try this search in The ACM Guide](#)

Display results

expanded form



[Search Tips](#)

☐ Open results in a new window

Results 1 - 20 of 200

Result page: [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) [9](#) [10](#) [next](#)

Best 200 shown

Relevance scale ☐ ☐ ☐ ☐ ☐

1 [Fast detection of communication patterns in distributed executions](#)

Thomas Kunz, Michiel F. H. Seuren

November 1997 **Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research**

Full text available: [pdf\(4.21 MB\)](#) Additional Information: [full citation](#), [abstract](#), [references](#), [index terms](#)

Understanding distributed applications is a tedious and difficult task. Visualizations based on process-time diagrams are often used to obtain a better understanding of the execution of the application. The visualization tool we use is Poet, an event tracer developed at the University of Waterloo. However, these diagrams are often very complex and do not provide the user with the desired overview of the application. In our experience, such tools display repeated occurrences of non-trivial commun ...

2 [Data flow analysis for checking properties of concurrent Java programs](#)

Gleb Naumovich, George S. Avrunin, Lori A. Clarke

May 1999 **Proceedings of the 21st international conference on Software engineering**

Full text available: [pdf\(1.43 MB\)](#) Additional Information: [full citation](#), [references](#), [citations](#), [index terms](#)

Keywords: Java, concurrency, data flow, static analysis

3 [Core semantics of multithreaded Java](#)

Jeremy Manson, William Pugh

June 2001 **Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande**

Full text available: [pdf\(816.27 KB\)](#) Additional Information: [full citation](#), [abstract](#), [references](#), [citations](#), [index terms](#)

Java has integrated multithreading to a far greater extent than most programming languages. It is also one of the only languages that specifies and requires safety guarantees for improperly synchronized programs. It turns out that understanding these issues is far more subtle and difficult than was previously thought. The existing specification makes guarantees that prohibit standard and proposed compiler optimizations; it also omits guarantees that are necessary for safe execution of much ex ...

4 [Part II: Articles: A side-by-side comparison of exception handling in Ada and Java](#)

Alfred Strohmeier, Stanislav Chachkov
September 2001 **ACM SIGAda Ada Letters**, Volume XXI Issue 3


Full text available:  pdf(877.81 KB) Additional Information: [full citation](#), [abstract](#), [references](#)

The purpose of this paper is to compare the exception handling mechanisms of Ada and Java. In order to be intelligible and useful to both communities, we have tried not to get into specific technical intricacies of the languages, perhaps sometimes at the cost of precision. Nevertheless, we decided to use the language-specific terminology whenever we write about a given language. We believe that the contrary would often lead to misunderstandings: a) the same term sometimes covers two different co ...

5 A study of locking objects with bimodal fields

Tamiya Onodera, Kiyokuni Kawachiya

October 1999 **ACM SIGPLAN Notices , Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications**, Volume 34 Issue 10

Full text available:  pdf(1.45 MB) Additional Information: [full citation](#), [abstract](#), [references](#), [citations](#), [index terms](#)

Object locking can be efficiently implemented by bimodal use of a field reserved in an object. The field is used as a lightweight lock in one mode, while it holds a reference to a heavyweight lock in the other mode. A bimodal locking algorithm recently proposed for Java achieves the highest performance in the absence of contention, and is still fast enough when contention occurs. However, mode transitions inherent in bimodal locking have not yet been fully considered. The algorithm ...

6 Termination in language-based systems

Algis Rudys, Dan S. Wallach

May 2002 **ACM Transactions on Information and System Security (TISSEC)**, Volume 5 Issue 2

Full text available:  pdf(355.43 KB) Additional Information: [full citation](#), [abstract](#), [references](#), [index terms](#)

Language run-time systems are increasingly being embedded in systems to support run-time extensibility via mobile code. Such systems raise a number of concerns when the code running in such systems is potentially buggy or untrusted. Although sophisticated access controls have been designed for mobile code and are shipping as part of commercial systems such as Java, there is no support for terminating mobile code short of terminating the entire language run-time. This article presents a c ...

Keywords: Applets, Internet, Java, resource control, soft termination, termination

7 An efficient algorithm for computing MHP information for concurrent Java programs

Gleb Naumovich, George S. Avrunin, Lori A. Clarke

October 1999 **ACM SIGSOFT Software Engineering Notes , Proceedings of the 7th European engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering**, Volume 24 Issue 6


Full text available:  pdf(1.32 MB) Additional Information: [full citation](#), [abstract](#), [references](#), [citations](#), [index terms](#)

Information about which statements in a concurrent program may happen in parallel (MHP) has a number of important applications. It can be used in program optimization, debugging, program understanding tools, improving the accuracy of data flow approaches, and detecting synchronization anomalies, such as data races. In this paper we propose a data flow algorithm for computing a conservative estimate of the MHP information for Java programs that has a worst-c ...

8 Executing Java threads in parallel in a distributed-memory environment

Mark W. MacBeth, Keith A. McGuigan, Philip J. Hatcher

November 1998 **Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research**

Full text available:  [pdf\(194.63 KB\)](#) Additional Information: [full citation](#), [abstract](#), [references](#), [index terms](#)

We present the design and initial implementation of Hyperion, an environment for the high-performance execution of Java programs. Hyperion supports high performance by utilizing a Java-bytecode-to-C translator and by supporting parallel execution via the distribution of Java threads across the multiple processors of a cluster of Linux machines. The Hyperion run-time system implements the Java memory model using an efficient communication substrate previously developed for Linux and Fast Ethernet ...

9 Part II: Articles: Implementing transactions using Ada exceptions: which features are missing?

M. Patiño-Martínez, R. Jiménez-Peris, S. Arévalo

September 2001 **ACM SIGAda Ada Letters**, Volume XXI Issue 3

Full text available:  [pdf\(924.53 KB\)](#) Additional Information: [full citation](#), [abstract](#), [references](#)

Transactional Drago programming language is an *Ada* extension that provides transaction processing capabilities. Exceptions have been integrated with transactions in *Transactional Drago*; exceptions are used to notify transaction aborts and any unhandled exception aborts a transaction. Transactions can be multithreaded in *Transactional Drago*, and therefore, concurrent exceptions can be raised. In that case a single exception must be chosen to notify the transaction abort ...

Keywords: ada 95, concurrent exception resolution, exceptions, transactions

10 Deterministic replay of Java multithreaded applications

Jong-Deok Choi, Harini Srinivasan

August 1998 **Proceedings of the SIGMETRICS symposium on Parallel and distributed tools**

Full text available:  [pdf\(1.45 MB\)](#) Additional Information: [full citation](#), [references](#), [citations](#), [index terms](#)

11 Specifying Java thread semantics using a uniform memory model

Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom

November 2002 **Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande**

Full text available:  [pdf\(202.03 KB\)](#) Additional Information: [full citation](#), [abstract](#), [references](#), [index terms](#)

Standardized language level support for threads is one of the most important features of Java. However, defining the Java Memory Model (JMM) has turned out to be a major challenge. Several models produced to date are not as easily comprehensible and comparable as first thought. Given the growing interest in multithreaded Java programming, it is essential to have a sound framework that would allow formal specification and reasoning about the JMM. This paper presents the Uniform Memory Model (UMM), ...

Keywords: Java, compilation, memory models, threads, verification

12 Portable resource control in Java

Walter Binder, Jane G. Hulaas, Alex Villazón

October 2001 **ACM SIGPLAN Notices , Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and**

applications, Volume 36 Issue 11Full text available:  pdf(307.08 KB)Additional Information: [full citation](#), [abstract](#), [references](#), [citations](#), [index terms](#)

Preventing abusive resource consumption is indispensable for all kinds of systems that execute untrusted mobile code, such as mobile object systems, extensible web servers, and web browsers. To implement the required defense mechanisms, some support for resource control must be available: accounting and limiting the usage of physical resources like CPU and memory, and of logical resources like threads. Java is the predominant implementation language for the kind of systems envisaged here, even though ...

Keywords: Java, bytecode rewriting, micro-kernels, mobile object systems, resource control, security

13 The apprentice challenge

J. Strother Moore, George Porter

May 2002 **ACM Transactions on Programming Languages and Systems (TOPLAS)**,

Volume 24 Issue 3


Full text available:  pdf(212.09 KB)Additional Information: [full citation](#), [abstract](#), [references](#), [citations](#), [index terms](#)

We describe a mechanically checked proof of a property of a small system of Java programs involving an unbounded number of threads and synchronization, via monitors. We adopt the output of the javac compiler as the semantics and verify the system at the bytecode level under an operational semantics for the JVM. We assume a sequentially consistent memory model and atomicity at the bytecode level. Our operational semantics is expressed in ACL2, a Lisp-based logic of recursive functions. Our proofs ...

Keywords: Java, Java Virtual Machine, mutual exclusion, operational semantics, parallel and distributed computation, theorem proving

14 Technical papers: concurrency: Assuring and evolving concurrent programs: annotations and policy

Aaron Greenhouse, William L. Scherlis

May 2002 **Proceedings of the 24th international conference on Software engineering**Full text available:  pdf(1.38 MB)Additional Information: [full citation](#), [abstract](#), [references](#), [citations](#), [index terms](#)

Assuring and evolving concurrent programs requires understanding the concurrency-related design decisions used in their implementation. In Java-style shared-memory programs, these decisions include which state is shared, how access to it is regulated, the roles of threads, and the policy that distinguishes desired concurrency from race conditions. These decisions rarely have purely local manifestations in code. In this paper, we use case studies from production Java code to explore the costs and ...

15 Efficient and precise data race detection for multithreaded object-oriented programs

Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, Manu Sridharan

May 2002 **ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation**, Volume 37 Issue 5Full text available:  pdf(171.13 KB)Additional Information: [full citation](#), [abstract](#), [references](#), [citations](#), [index terms](#)

We present a novel approach to dynamic data race detection for multithreaded object-oriented programs. Past techniques for on-the-fly data race detection either sacrificed precision for performance, leading to many false positive data race reports, or maintained

precision but incurred significant overheads in the range of 3x to 30x. In contrast, our approach results in very few false positives and runtime overhead in the 13% to 42% range, making it both efficient *and* precis ...

Keywords: dataraces, debugging, multithreaded programming, object-oriented programming, parallel programs, race conditions, static-dynamic co-analysis, synchronization

16 Sapphire: copying GC without stopping the world

Richard L. Hudson, J. Eliot B. Moss

June 2001 **Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande**

Full text available:  pdf(899.45 KB) Additional Information: [full citation](#), [abstract](#), [references](#), [citations](#), [index terms](#)

Many concurrent garbage collection (GC) algorithms have been devised, but few have been implemented and evaluated, particularly for the Java programming language. Sapphire is an algorithm we have devised for concurrent copying GC. Sapphire stresses minimizing the amount of time any given application thread may need to block to support the collector. In particular, Sapphire is intended to work well in the presence of a large number of application threads, on small- to medium-scale shared memory ...

17 Specification and implementation of exceptions in workflow management systems

Fabio Casati, Stefano Ceri, Stefano Paraboschi, Giuseppe Pozzi

September 1999 **ACM Transactions on Database Systems (TODS)**, Volume 24 Issue 3

Full text available:  pdf(250.40 KB) Additional Information: [full citation](#), [abstract](#), [references](#), [citations](#), [index terms](#)

Although workflow management systems are most applicable when an organization follows standard business processes and routines, any of these processes faces the need for handling exceptions, i.e., asynchronous and anomalous situations that fall outside the normal control flow. In this paper we concentrate upon anomalous situations that, although unusual, are part of the semantics of workflow applications, and should be specified and monitored coherently; in most real-life applications ...

Keywords: active rules, asynchronous events, exceptions, workflow management systems

18 Implementing an on-the-fly garbage collector for Java

Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Eliot E. Salant, Katherine Barabash, Itai Lahan, Yossi Levanoni, Erez Petrank, Igor Yanorer

October 2000 **ACM SIGPLAN Notices , Proceedings of the second international symposium on Memory management**, Volume 36 Issue 1

Full text available:  pdf(1.33 MB) Additional Information: [full citation](#), [abstract](#), [citations](#), [index terms](#)

Java uses garbage collection (GC) for the automatic reclamation of computer memory no longer required by a running application. GC implementations for Java Virtual Machines (JVM) are typically designed for single processor machines, and do not necessarily perform well for a server program with many threads running on a multiprocessor. We designed and implemented an on-the-fly GC, based on the algorithm of Doligez, Leroy and Gonthier [13, 12] (DLG), for Java in this environment. An *on-the-fly* ...

Keywords: Java, concurrent garbage collection, garbage collection, memory management, on-the-fly garbage collection, programming languages

Multitasking without compromise: a virtual machine evolution

Grzegorz Czajkowski, Laurent Daynés

October 2001 **ACM SIGPLAN Notices , Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications**, Volume 36 Issue 11

Full text available:  pdf(220.97 KB) Additional Information: [full citation](#), [abstract](#), [references](#), [citations](#), [index terms](#)

The multitasking virtual machine (called from now on simply MVM) is a modification of the Java virtual machine. It enables safe, secure, and scalable multitasking. Safety is achieved by strict isolation of application from one another. Resource control augment security by preventing some denial-of-service attacks. Improved scalability results from an aggressive application of the main design principle of MVM: share as much of the runtime as possible among applications and replicate everything el ...

Keywords: Java virtual machine, application isolation, native code execution, resource control

20 Eliminating synchronization bottlenecks using adaptive replication

Martin C. Rinard, Pedro C. Diniz

May 2003 **ACM Transactions on Programming Languages and Systems (TOPLAS)**, Volume 25 Issue 3

Full text available:  pdf(826.28 KB) Additional Information: [full citation](#), [abstract](#), [references](#), [index terms](#)

This article presents a new technique, adaptive replication, for automatically eliminating synchronization bottlenecks in multithreaded programs that perform atomic operations on objects. Synchronization bottlenecks occur when multiple threads attempt to concurrently update the same object. It is often possible to eliminate synchronization bottlenecks by replicating objects. Each thread can then update its own local replica without synchronization and without interacting with other threads. When ...

Keywords: Atomic operations, commutativity analysis, parallel computing, parallelizing compilers, replication, synchronization

Results 1 - 20 of 200

Result page: [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) [9](#) [10](#) [next](#)

The ACM Portal is published by the Association for Computing Machinery. Copyright © 2004 ACM, Inc.

[Terms of Usage](#) [Privacy Policy](#) [Code of Ethics](#) [Contact Us](#)

Useful downloads:  [Adobe Acrobat](#)  [QuickTime](#)  [Windows Media Player](#)  [Real Player](#)

[First Hit](#) [Fwd Refs](#)**End of Result Set**☐ [Generate Collection](#) [Print](#)

L15: Entry 1 of 1

File: USPT

Mar 23, 2004

DOCUMENT-IDENTIFIER: US 6711739 B1

TITLE: System and method for handling threads of execution

Abstract Text (1):

A mechanism for controlling threads in a Java application while avoiding the unsafe conditions inherent in the use of existing java.lang.Thread methods. In one embodiment, a first class is defined for handling threads in an application. The first class uses a target variable to indicate whether a thread should continue to run, or whether it should be stopped. This first class provides a start() method to set up the target variable, a stop() method to set the target variable to indicate that the thread should be stopped, and an abstract run() method. The functionality of the run() method is provided by one or more additional classes which extend the first class. The additional classes override the abstract run() method and define the tasks to be performed by threaded objects instantiated from these classes. When a thread needs to be stopped, the corresponding target variable is set to indicate that it should be stopped. The thread periodically checks the target variable and, when the target variable is set to indicate that the thread should be stopped, the thread executes one or more instructions that cause execution of the thread to complete and to exit normally.

Brief Summary Text (7):

Threads can be implemented in Java by creating a subclass of java.lang.Thread, or by using the java.lang.Runnable interface. The java.lang.Thread class includes three methods which are the primary means for controlling threads: start(), run() and stop(). The start() method prepares the thread to be executed. The run() method performs the functions of the thread. The stop() method terminates the thread. The java.lang.Thread class also includes several other methods which are used to control the execution of instructions in a thread, including suspends, resumes, sleep() and yield().

Brief Summary Text (8):

Two of these methods are inherently problematic. It may be unsafe to use the stop() method because, when a thread is terminated, objects which may have been locked by the thread are all unlocked, regardless of whether or not they were in consistent states. These inconsistencies can lead to arbitrary behavior which make corrupt the functions of the thread and the program. The suspend() method may also cause problems because it is prone to deadlock. For example, if a thread holds a lock on the monitor for a particular resource when it is suspended, no other thread can access the resource until the thread holding the lock is resumed. If a second thread should cause the first thread to resume, but first attempts to access the resource, it may wait for access to the resource and consequently may never call the first thread's resume() method. Because these two methods are so problematic, their use is discouraged and they are being deprecated from the Java Developers Kit produced by Java's originator, Sun Microsystems Inc.

Brief Summary Text (12):

When an object is instantiated from the subclass, the start() method inherited from the class is configured to create a thread having the object as its target.

The start() method is also configured to set the target variable (which is local to the thread) is set to a value which indicates that the thread should be running. The stop() method of the class is also inherited by the subclass. When the stop() method is invoked, it is configured to set the target variable to a value which indicates that the thread should be stopped. The run() method provided by the subclass periodically checks the target variable within the thread. The checking of the target variable occurs in the normal course of execution of the run method. If the target variable indicates that the thread should be stopped, the run() method is configured to complete execution and exit normally, causing the thread to terminate. An exception is not required to stop the run() method, so exception handling is not necessary.

Brief Summary Text (14):

In one embodiment, a thread-handling method is provided for improved handling of threads in Java applications. Broadly speaking, the method comprises providing a class that includes methods for stopping threads based on the indication of a target variable. Instructions that are to be executed within threads are provided in the run() methods of subclasses that extend the first class. Rather than creating threads from the standard Thread class and individually configuring the threads to stop execution upon the occurrence of a particular condition, threads are created using the subclasses above. The safer methods which are inherited from these subclasses override the methods of the Thread class so that stopping threads is inherently safer. When a thread is created, the target variable is initialized to indicate that the thread should be running. The instructions being executed by the thread periodically check the target variable to determine whether it indicates that the thread should continue running, or should stop. If the target variable indicates that the thread should continue running, the thread executes normally until the target variable is checked again. If the target variable indicates that the thread should be stopped, the run() method completes execution and exits normally.

Detailed Description Text (3):

One embodiment of the invention is described below. In this embodiment, a class is written to handle the threads which may be used in an application. (For the purposes of this disclosure, this class will be referred to as the "Handler" class.) The threads are created using this class. When a thread needs to be stopped, a target variable defined in the class is set to indicate that the thread should be stopped. The thread periodically checks the target variable to determine whether or not it should stop. The target variable is checked by the thread when it is in a stable state so that, if the thread should be stopped, the run() method can complete execution and terminate normally rather than having to use the stop() method defined in the java.lang.Thread class.

Detailed Description Text (4):

Before describing the present mechanism, it may be helpful to describe the operation of a thread. As indicated above, a thread is a separate stream of execution that can take place independently from and concurrently with other streams of execution. A thread is created, then it runs, and then it dies. Execution of instructions within a thread may be suspended, or the thread may be put to sleep, blocked or made to yield to other threads, after which execution may continue. A thread may be terminated by completing its run() method, by another thread preempting it or by calling its stop() method.

Detailed Description Text (22):

Threads which are in the "not-runnable" state, on the other hand, will not be executed even if a processor is available to execute them. Several events may cause the threads to be not-runnable. For example, if a thread's sleep() method is invoked, that thread will wait for a designated amount of time before continuing execution. In another instance, if a thread's suspend() method is invoked, execution of the thread will be discontinued until its resume() method is called.

A thread may also be in the not-runnable. state if it is blocked while waiting for I/O. It should be noted that, when these threads are once again runnable, they may not actually begin execution if a processor is not available.

Detailed Description Text (23):

If a thread is stopped, it is in the "dead" state. A dead thread cannot resume execution in the same manner as a not-runnable thread, but it can be re-started. Execution of a thread may be stopped in several ways. Preferably, a thread stops when execution of its run() method has completed (i.e., if there are no more instructions to be executed.) There may also be instances in which it may be desirable to have a thread stop on demand. For example, it may be desirable to have a thread continue execution indefinitely and to stop when instructed to do so. The stop() method is provided for this purpose in the java.lang.Thread class. The Thread.stop() method, however, may cause the thread to be terminated when it is in an unstable state. This problem may be illustrated using FIG. 2.

Detailed Description Text (24):

Referring to FIG. 2, a flow diagram illustrating the execution of instructions in a thread is shown. In this example, the thread is configured to execute instructions 1-N, then loop back and repeat these instructions. The thread will continue to execute instructions 1-N until the thread is stopped. If the Thread.stop() method is invoked to stop the thread, it will immediately cause an exception and terminate the thread's execution. The stop() method is not constrained to halt execution of the thread at any particular point, so there is no way to determine where among these instructions execution will be stopped. The thread may be stopped at point A, point B, point C, or any other point in the execution of the thread's run() method. If the instructions modify the state of the application, stopping the thread may leave the application in an unknown state. The thread will also unlock all of the monitors which had been locked by the thread, possibly leaving objects protected by these monitors in inconsistent states. Objects which are left in these inconsistent states are said to be damaged, and operations on these damaged objects may lead to arbitrary behavior. Errors that are caused by the behavior of damaged objects may be difficult to detect and a user may have no warning that the errors have occurred.

Detailed Description Text (25):

It is therefore preferable to stop a thread by allowing it to complete the run() method. The present mechanism employs a target variable associated with the thread to provide a way to instruct the thread to stop, while allowing the run() method to complete execution. Essentially, the target variable provides an indication of whether the thread should continue to run or to stop. The thread periodically checks the target variable. If the target variable indicates that the thread should continue to run, execution of the thread proceeds normally. If the target variable indicates that the thread should stop, execution of the thread proceeds to the point at which the target variable is checked, then exits normally (i.e., completes the run() method.)

Detailed Description Text (26):

Referring to FIG. 3, a flow diagram illustrating the execution of instructions in a thread using the present mechanism is shown. After the start() method is called, the body of the run() method is executed. The flow diagram on the left side of the figure represents body of the run() method. The thread is still configured to execute instructions 1-N, but it is further configured to periodically examine the target variable to determine its value (e.g., by using the isRunning() method described below.) If the target variable is set to indicate that the thread should continue running, instructions 1-N are repeated. If the target variable is set to indicate that the thread should be stopped (e.g., using the stop() method illustrated here as a different thread of execution,) the thread branches to point A, where it completes execution and exits normally. Because instructions 1-N are completed normally before the target variable is checked, the state of the

application is easier to determine. Because the run() method executes to completion, no exception handling is required and no objects are damaged by abnormal termination of the thread. It should be noted that the target variable can be checked at different points in the code of the run() method. It should also be noted that the target variable can be set by other threads or by the run() method itself to indicate that the thread should be stopped.

Detailed Description Text (50):

The stop() method of the Handler class provides a means for gracefully terminating the thread and should not be overridden by the second class. By providing an indication that the thread should be stopped rather than simply stopping the thread using the stop() method of the thread class, the instability inherent in the Thread.stop() method is avoided. The thread does not immediately throw an exception (unlocking monitors as the exception propagates up the stack,) but instead allows the thread to terminate normally, leaving the system in a stable state.

Detailed Description Text (53):

The Handler class described above thereby provides the following advantages: first, it gracefully handles stopping a thread without exception handling; second, classes which extend the Handler class implement the Runnable interface and can be referenced as Runnable; third, it can be determined locally within a thread whether the thread should continue to run, or should be terminated, based on the result of the isRunning() method; and fourth, because this class has such a simple API, it provides a suitable core for Java servers, agents and socket handlers to handle asynchronous peer communications. The Handler class thereby provides a means for standardization in the handling of threads which is object oriented, which uses self-contained logic, and which allows developers to use programming techniques with which they are already familiar. (It should be noted that other embodiments may vary from the implementation described above and may therefore provide advantages which differ from those listed here.)

CLAIMS:

8. The system of claim 1, wherein each of the threads of the Java application is configured to leave the Java application in a known state after termination of execution of the thread.

17. A system, comprising: a dedicated Java thread handler class which includes: a start method that sets a target variable to indicate that threads extending said dedicated Java thread handler class are running, an abstract run method, and a stop method that sets said target variable to indicate that said threads extending said Java class are to be terminated; wherein said dedicated Java thread handler class is configured to be extended by custom thread subclasses each including a run method that overrides said abstract run method, wherein said dedicated Java thread handler class does not include functionality particular to any of the custom thread subclasses, wherein each run method provides code to be executed by a thread implementing a particular one of the subclasses including said run method; and wherein, during execution of said thread, if said target variable is set to indicate that said thread is to be terminated, said run method is configured to complete execution to terminate the thread normally.

18. The system of claim 17, wherein, to terminate the thread normally, no exceptions are thrown and one or more objects accessed by said thread are left in a consistent state.

19. The system of claim 17, wherein, to terminate the thread normally, a Java application in which the thread is configured to execute is left in a known state.

21. The system of claim 20, wherein each of the threads of the Java application is

configured to leave the Java application in a known state after termination of execution of the thread.

First Hit Fwd Refs
End of Result Set

☐ **Generate Collection** **Print**

L30: Entry 2 of 2

File: USPT

Jun 27, 2000

DOCUMENT-IDENTIFIER: US 6081665 A

TITLE: Method for efficient soft real-time execution of portable byte code computer programs

Abstract Text (1):

The invention is a method for use in executing portable virtual machine computer programs under real-time constraints. The invention includes a method for implementing a single abstract virtual machine execution stack with multiple independent stacks in order to improve the efficiency of distinguishing memory pointers from non-pointers. Further, the invention includes a method for rewriting certain of the virtual machine instructions into a new instruction set that more efficiently manipulates the multiple stacks. Additionally, using the multiple-stack technique to identify pointers on the run-time stack, the invention includes a method for performing efficient defragmenting real-time garbage collection using a mostly stationary technique. The invention also includes a method for efficiently mixing a combination of byte-code, native, and JIT-translated methods in the implementation of a particular task, where byte-code methods are represented in the instruction set of the virtual machine, native methods are written in a language like C and represented by native machine code, and JIT-translated methods result from automatic translation of byte-code methods into the native machine code of the host machine. Also included in the invention is a method to implement a real-time task dispatcher that supports arbitrary numbers of real-time task priorities given an underlying real-time operating system that supports at least three task priority levels. Finally, the invention includes a method to analyze and preconfigure virtual memory programs so that they can be stored in ROM memory prior to program.

Brief Summary Text (4):

Java (a trademark of Sun Microsystems, Inc.) is an object-oriented programming language with syntax derived from C and C++. However, Java's designers chose not to pursue full compatibility with C and C++ because they preferred to eliminate from these languages what they considered to be troublesome features. In particular, Java does not support enumerated constants, pointer arithmetic, traditional functions, structures and unions, multiple inheritance, goto statements, operator overloading, and preprocessor directives. In their place, Java requires all constant identifiers, functions (methods), and structures to be encapsulated within

Brief Summary Text (5):

class (object) declarations. The purpose of this requirement is to reduce conflicts in the global name space. Java provides standardized support for multiple threads (lightweight tasks) and automatic garbage collection of dynamically-allocated memory. Furthermore, Java fully specifies the behavior of every operator on every type, unlike C and C++ which leave many behaviors to be implementation dependent. These changes were designed to improve software scalability, reduce software development and maintenance costs, and to achieve full portability of Java software. Anecdotal evidence suggests that many former C and C++ programmers have welcomed these language improvements.

Brief Summary Text (6):

One distinguishing characteristic of Java is its execution model. Java programs are first translated into a fully portable standard byte code representation. The byte code is then available for execution on any Java virtual machine. A Java virtual machine is simply any software system that is capable of understanding and executing the standard Java byte code representation. Java virtual machine support is currently available for AIX, Apple Macintosh, HPUX, Linux, Microsoft NT, Microsoft Windows 3.1, Microsoft Windows 95, MVS, Silicon Graphics IRIX, and Sun Solaris. Ports to other environments are currently in progress. To prevent viruses from being introduced into a computer by a foreign Java byte-code program, the Java virtual machine includes a Java byte code analyzer that verifies the byte code does not contain requests that would compromise the local system. By convention, this byte code analyzer is applied to every Java program before it is executed. Byte code analysis is combined with optional run-time restrictions on access to the local file system for even greater security. Current Java implementations use interpreters to execute the byte codes but future high-performance Java systems will have the capability of translating byte codes to native machine code on the fly. In theory, this will allow Java programs to run approximately at the same speed as C++.

Brief Summary Text (7):

Within Sun, development of Java began in April of 1991. Initially, Java was intended to be an implementation language for personal digital assistants. Subsequently, the development effort was retargeted to the needs of set-top boxes, CD-ROM software, and ultimately the World-Wide Web. Most of Java's recent media attention has focused on its use as a medium for portable distribution of software over the Internet. However, both within and outside of Sun, it is well understood that Java is much more than simply a language for adding animations to Web pages. In many embedded real-time applications, for example, the Java byte codes might be represented in system ROMs or might even be pre-translated into native machine code.

Brief Summary Text (8):

Many of the more ambitious "industrial-strength" sorts of applications that Java promises to enable on the Internet have associated real-time constraints. These applications include video conferencing integrated with distributed white boards, virtual reality, voice processing, full-motion video and real-time audio for instruction and entertainment, and distributed video games. More importantly, the next generation Web client will have even more real-time requirements. Future set-top devices will connect home televisions to the Web by way of cable TV networks. Besides all of the capabilities just mentioned, these systems will also support fully interactive television applications.

Brief Summary Text (9):

Java offers important software engineering benefits over C and C++, two of the more popular languages for current implementation of embedded real-time systems. If Java could be extended in ways that would allow it to support the cost-effective creation of portable, reliable real-time applications, the benefits of this programming language would be realized by a much larger audience than just the people who are implementing real-time Web applications. All developers of embedded real-time software could benefit. Some of the near-term applications for which a real-time dialect of Java would be especially well suited include personal digital assistants, real-time digital diagnosis (medical instrumentation, automotive repair, electronics equipment), robotics, weather monitoring and forecasting, emergency and service vehicle dispatch systems, in-vehicle navigation systems, home and business security systems, military surveillance, radar and sonar analysis, air traffic control, and various telephone and Internet packet switching applications.

Brief Summary Text (10):

This invention relates generally to computer programming methods pertaining to

real-time applications and more specifically to programming language implementation methods which enable development of real-time software that can run on computer systems of different designs. PERC (a trademark of NewMonics Inc.) is a dialect of the Java programming language designed to address the special needs of developers of real-time software.

Brief Summary Text (12):

Unlike many existing real-time systems, most of the applications for which PERC is intended are highly dynamic. New real-time workloads arrive continually and must be integrated into the existing workload. This requires dynamic management of memory and on-the-fly schedulability analysis. Price and performance issues are very important, making certain traditional real-time methodologies cost prohibitive. An additional complication is that an application developer is not able to test the software in each environment in which it is expected to run. The same Java byte-code application would have to run within the same real-time constraints on a 50 MHz 486 and on a 300 MHz Digital Alpha. Furthermore, each execution environment is likely to have a different mix of competing applications with which this code must contend for CPU and memory resources. Finally, every Java byte-code program is supposed to run on every Java virtual machine, even a virtual machine that is running as one of many tasks executing on a time-sharing host. Clearly, time-shared virtual machines are not able to offer the same real-time predictability as a specially designed PERC virtual machine embedded within a dedicated microprocessor environment. Nevertheless, such systems are able to provide soft-real-time response.

Brief Summary Text (26):

Java, a trademark of Sun Microsystems, Inc., is an object-oriented programming language with syntax derived from C and C++, which provides automatic garbage collection and multi-threading support as part of the standard language definition.

Brief Summary Text (27):

JIT, as the term is used in this invention disclosure, is an acronym standing for "just in time." The term is used to describe a system for translating Java byte codes to native machine language on the fly, just-in-time for its execution. We consider any translation of byte code to machine language which is carried out by the virtual machine to be a form of JIT compilation.

Brief Summary Text (30):

Native Method, as this term is used in relation to the Java and PERC programming languages, describes a method that is implemented in C (or some other low-level language) rather than in the high-level Java or PERC language in which the majority of methods are implemented.

Brief Summary Text (31):

PERC, a trademark of NewMonics Inc., is an object-oriented programming language with similarities to Java, which has been designed to address the specific needs of developers of real-time and embedded software.

Brief Summary Text (43):

Thread is a term of art describing a computer program that executes with an independent flow of execution. Java is a threaded language, meaning that multiple flows of execution may be active concurrently. All threads share access to the same global memory pool. (In other programming environments, threads are known as tasks.)

Brief Summary Text (49):

1. Extensions to the standard Java byte code instruction set to enable efficient run-time isolation of pointer variables from non-pointer variables. The extended byte codes are described as the PERC instruction set.

Brief Summary Text (50):

2. A mechanism to translate traditional Java byte codes into the extended PERC byte codes at run-time, as new Java byte codes are loaded into the virtual machine's execution environment.

Brief Summary Text (52):

of the PERC instruction set. The Java run-time stack is replaced by two stacks, one for non-pointer and the other for pointer data. Further, the data structures enable efficient interaction between native methods, Java methods represented by byte code, and Java methods translated by a JIT compiler to native machine language. Performance tradeoffs are biased to give favorable treatment to execution of JIT-translated methods.

Brief Summary Text (56):

7. A mechanism for translating traditional Java byte codes into the extended PERC byte codes prior to run-time, in order to reduce run-time overhead and simplify system organization.

Drawing Description Text (6):

FIG. 5 illustrates the appearance of the pointer and non-pointer stack activation frames immediately before calling and immediately following entry into the body of a Java method. The stacks are assumed to grow downward. In preparation for the call, arguments are pushed onto the stack. Within the called method, the frame pointer (fp) is adjusted to point at the memory immediately above the first pushed argument and the stack pointer (sp) is adjusted to make room for local variables to be stored on the stack.

Drawing Description Text (7):

FIG. 6 illustrates the internal organization of the local-variable region of the stack activation frame. This region includes application-declared locals (as declared in byte-code attributes for Java methods and as specified in the parameterization of BuildFrames(), temporary variables (as might be required to represent the old values of the frame and instruction pointers), a run-time stack (to allow execution of push and pop operations within the method), and space for arguments to be pushed to other methods to be called from this method.

Drawing Description Text (14):

FIG. 13 provides C preprocessor definitions of symbolic constants used to describe the encodings of Sun's Java byte code instruction set.

Drawing Description Text (21):

FIG. 20 provides the C declaration of the structure used internal to the PERC implementation to represent a HashLock structure. Exactly one HashLock structure is allocated for each PERC object that needs either a hash value or a lock, or both.

Drawing Description Text (44):

FIG. 43 provides C macros used by application code to coordinate with the dispatcher. The application executes the CheckPreemption() macro to see if the dispatcher wants it to preempt itself. The application executes PreemptTask() when the task is ready to be preempted. The application executes PrepareBlockCall() immediately before calling a system routine which may block. It executes ResumeAfterBlockCall() upon return from the system routine.

Drawing Description Text (54):

FIG. 53 provides the Java implementation of the TaskDispatcher class .

Drawing Description Text (70):

FIG. 69 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the IADD instruction, which adds the

two integers on the top of the Java stack, placing the result on the Java stack.

Drawing Description Text (91):

FIG. 90 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the MONITORENTER instruction.

Drawing Description Text (92):

FIG. 91 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the MONITOREXIT instruction.

Drawing Description Text (97):

FIG. 96 illustrates the Java implementation of the Atomic class for use on uniprocessor systems that lack the capability to analyze worst-case execution times. Application programmers can prevent threads from being preempted within certain critical regions by surrounding those regions with execution of Atomic.enter() and Atomic.exit().

Detailed Description Text (4):

PERC (and Java) is an object-oriented programming language. Programs are comprised of object type declarations, known in PERC as classes. Each class definition describes the variables that are associated with each instance (object) of the corresponding class and also defines all of the operations that can be applied to instantiated objects of this type. Operations are known as methods.

Detailed Description Text (18):

PERC, like Java, supports four distinct forms of method invocation. These are known as (1) virtual, (2) special (non-virtual), (3) static, and (4) interface. With virtual and special method invocations, there is an implicit (not seen by the Java programmer) "this" argument passed to the called method. The "this" argument refers to the object on which the called method will operate. The distinctions between these different method invocations are described in "The Java Virtual Machine Specification", by Lindholm and Yellin, 1996, Addison-Wesley.

Detailed Description Text (20):

The PERC implementation represents every PERC object with a data structure patterned after the templates provided in FIG. 15, FIG. 16, and FIG. 24. In all of these structures, the second field is a pointer to a MethodTable data structure (see FIG. 23). The PERC execution environment maintains one MethodTable data structure for each defined object type. All instantiated objects of this type point to this shared single copy. The jit.sub.-- interfaces array field of the MethodTable structure has one entry for each virtual method supported by objects of this type. The mapping from method name and signature to index position is defined by the class loader, as described in "The Java Virtual Machine Specification", by Lindholm and Yellin, 1996, Addison-Wesley. To execute the JIT version of a PERC method using a virtual method lookup, branch to the code represented by jit.sub.-- interfaces[method.sub.-- index]. Normally, the JIT version of the byte code will only be invoked directly from within another JIT-compiled method. If a native or untranslated byte-code method desires to invoke another method using virtual method lookup, the search for the target method generally proceeds differently. First, we find the target object's MethodTable data structure (as above) and then follow the methods pointer to obtain an array of pointers to Method objects. Within the Method object, we consult the access.sub.-- flags field to determine if the target method is represented by native code (ACC.sub.-- NATIVE) or JIT translation of byte code (ACC.sub.-- JIT). If neither of these flags is set, the method is assumed to be implemented by byte codes. See FIG. 49, FIG. 45, and FIG. 46.

Detailed Description Text (22):

When the method to be invoked by a particular operation is known at compile time, the Java compiler treats this as an invokeSpecial instruction. In these cases, there is no need to consult the method table at run time. When performing special

method invocation from within a JIT-translated method, the address of the called method (or at least a stub for the called method) is hard-coded into the caller's machine code.

Detailed Description Text (25):

When the method to be invoked is declared as static within the corresponding object (meaning that the method operates on class information rather than manipulating variables associated with a particular instance of the corresponding class), the Java compiler treats this as an invokeStatic method. Execution of static methods is identical to execution of special methods except that there is no implicit pointer to "this" passed as an argument to the called method. See FIG. 47, FIG. 45, and FIG. 46.

Detailed Description Text (59):

When JIT-translated code invokes a method that is implemented by Java byte code, it is necessary to switch the execution protocol prior to invoking pvm(). Rather than requiring JIT-generated code to check whether this protocol switch is necessary prior to each method invocation, we provide each byte-code method with a stub procedure that honors the JIT execution protocol. This stub procedure switches from JIT to C protocols and then invokes pvm() with appropriate arguments. In more detail, the stub procedure performs the following:

Detailed Description Text (64):

5. If the method to be invoked is synchronized, we enter the monitor now, waiting for other threads to exit first if necessary.

Detailed Description Text (68):

9. If the invoked method was synchronized, release the monitor now. Note that the pvm() itself takes responsibility for exiting the monitor if the code is aborted by throwing of an exception.

Detailed Description Text (224):

5.6 Support for Try Clauses (and monitors)

Detailed Description Text (228):

Native methods. Upon entry into a block of code that represents either a try block or a synchronized block, we save the previous value of the exception handling jump buffer in a local variable and set a stack-allocated jump buffer to represent this block's exception handler. Whenever an exception is raised, it performs a longjmp to the currently active exception handler. When this exception handler catches the longjmp invocation, it handles it if possible. Otherwise, it simply forwards the exception to the outer nested exception handling context.

Detailed Description Text (229):

When control leaves the exception handled block, we automatically restore the current exception handler to the value it held before this block was entered.

Detailed Description Text (230):

The PERC virtual machine. The PERC interpreter (virtual machine) is invoked once for each method to be interpreted. If the method to be interpreted contains synchronized or try blocks, a jump buffer is initialized according to the same protocol described above.

Detailed Description Text (273):

The PERC Virtual Machine describes the C function that interprets Java byte codes. This C function, illustrated in FIG. 68, is named pvm(). The single argument to pvm() is a pointer to a Method structure, which includes a pointer to the byte-code that represents the method's functionality. Each invocation of pvm() executes only a single method. To call another byte-code method, pvm() recursively calls itself. Note that pvm() is reentrant. When multiple Java threads are executing, each thread

executes byte-code methods by invoking pvm() on the thread's run-time stack.

Detailed Description Text (274):

The implementation of pvm() allocates space on the PERC pointer stack for three pointer variables. These pointers, known by the symbolic names pMETHOD, pBYTECODEF, and pCONSTANTS, represent pointers to the method's Method structure, the StringOfBytes object representing its byte code, and the constant-pool object representing the method's constant table respectively. During normal execution of pvm(), the values of these variables are stored in the C locals method, bytecode, and cp respectively. Before preemption, and before calling preemptible functions, pvm() copies the contents of these C variables onto the PERC pointer stack. In preparation for executing the byte codes representing a byte-code method, pvm() checks to determine if the method has any exception handlers. If the method is synchronized, the lock will have been obtained by the fastInvoke routine prior to calling pvm() (see FIG. 46). However, fastInvoke() does not set an exception handler to release the lock if the code is aborted by the raising of an exception. For this reason, pvm() sets an exception handler if the method is synchronized, so that it can release the lock before rethrowing the exception to the surrounding context.

Detailed Description Text (281):

5. If no exception handler is found, pvm() first releases the monitor lock if this method was synchronized and then it rethrows the exception to the surrounding exception handling context.

Detailed Description Text (303):

The MONITORENTER instruction (See FIG. 90) removes the object reference on the top of the pointer stack and arranges to apply a semaphore-like lock on that object. If the object is already locked by another thread, the current thread is put to sleep until the object becomes unlocked. Note that the pvm()'s state is saved and restored surrounding the call to the enterMonitor() function, because that call may result in preemption of this thread. Note that if the entry on the top of the pointer stack is NULL, this instruction throws an exception. The MONITOREXIT instruction (See FIG. 91) removes the object reference on the top of the pointer stack and arranges to remove its semaphore-like lock on that object. If the object has been locked multiple times by this thread, this instruction simply decrements the count on how many times this object has been locked rather than removing the lock. As with the MONITORENTER instruction, pvm()'s state is saved and restored surrounding the call to the exitMonitor() function and this instruction throws an exception if the top of the pointer stack is NULL.

Detailed Description Text (312):

6.2 Hash Values and Monitors

Detailed Description Text (313):

In concept, every Java object has an associated lock and an associated hash value. However, in practice, the large majority of Java objects never make use of either the lock or the hash value. Note that in systems that never relocate objects, converting an object's address to an integer value is probably the easiest way to obtain a hash value. However, in systems that make use of defragmenting garbage collectors, such as in the PERC execution environment, it is necessary to use some other technique to represent hash values.

Detailed Description Text (314):

In the PERC implementation, every object has a HashLock pointer field, which is initialized to NULL. When either a lock or a hash value is needed for the object, a HashLock object (see FIG. 20) is allocated and initialized, and the HashLock pointer field is made to refer to this HashLock object. Note that each HashLock object has the following fields:

Detailed Description Text (316):

2. The u field is a union which can represent either a pointer to another HashLock object (in case this HashLock object is currently residing on a free list), or a pointer to the thread that owns this semaphore if the lock is currently set, or NULL if this object is not currently on a free list and the lock is not currently set.

Detailed Description Text (317):

3. In case this semaphore is currently locked, waiting.sub.-- list points to a linked list of threads that are waiting for access to the locked object. The list is maintained in priority order.

Detailed Description Text (318):

4. In case this semaphore is currently locked, count represents the number of times the lock-holding thread has redundantly placed its lock on the corresponding object. The semaphore will not be released until this thread has removed its lock this many times. If the semaphore is not currently locked, count is zero.

Detailed Description Text (319):

Obtaining a hash value. When application code desires to obtain the hash value of a particular object, it invokes the native hashCode() method. This method consults the object's lock field. If this field is NULL, this method allocates a HashLock object, initializes its hash.sub.-- value field to the next available hash value, and initializes the object's lock pointer to refer to the newly allocated HashLock object. Then it returns the contents of the hash.sub.-- value field. If the lock field is non-NULL, hashCode() consults the hash.sub.-- value field of the corresponding HashLock object to determine whether a hash value has already been assigned. If this field has value 0, hashCode() overwrites the field with the next available hash value. Otherwise, the hash value has already been assigned. In all cases, the last step of hashCode() is to return the value of the hash.sub.-- value field.

Detailed Description Text (321):

Obtaining and releasing monitor locks. When application code desires to enter a monitor, it executes the enterMonitor instruction. This instruction first consults the object's lock field. If this field is NULL, it allocates a HashLock object, initializes its count field to 1, sets its u.owner field to represent the current thread, and grants access to the newly locked object. If the lock field is non-NULL, enterMonitor examines the contents of the HashLock object to determine whether access to the lock can be granted. If the count field equals 0, or if the u.owner field refers to the currently executing thread, the count field is incremented, the u.owner field is made to point to the current thread if it doesn't already, and access is granted to the newly locked object. Otherwise, this lock is owned by another thread. The current thread is placed onto the waiting.sub.-- list queue and its execution is blocked until the object's lock can be granted to this thread. Priority inheritance describes the notion that if a high-priority thread is forced to block waiting for a low-priority thread to release its lock on a particular object, the low-priority thread should temporarily inherit the priority of the higher priority blocked task. This is because, under this circumstance, the urgency of the locking task is increased by the fact that a high-priority task needs this task to get out of its way. The PERC virtual machine implements priority inheritance. Furthermore, the waiting.sub.-- list queue is maintained in priority order.

Detailed Description Text (322):

When a thread leaves a monitor, it releases the corresponding lock. This consists of the following steps:

Detailed Description Text (323):

1. Verifying that the monitor's u.owner field is the same as the currently

executing thread. Otherwise, this is an invalid request to exit the monitor.

Detailed Description Text (325):

3. If the waiting.sub.-- list queue is not empty, remove the leading (highest priority) entry from the queue. Make this the new u.owner of the lock and set the count field to 1. This is all that must be done. Stop. If the waiting.sub.-- list queue is empty, continue with step 4.

Detailed Description Text (326):

4. Otherwise, there is no longer a need to maintain this lock. Set the u.owner field to NULL.

Detailed Description Text (328):

6. Set the corresponding object's lock field to NULL and place this HashLock object onto a list of available HashLock objects, threaded through the u.next field. Whenever a new HashLock object is required, allocate from this free list if possible. Otherwise, allocate and initialize a new dynamic object.

Detailed Description Text (342):

b. An interrupt trigger may arrive, indicating that it is necessary to preempt the currently executing task so that a sporadic task can be executed (Of course, the dispatcher takes responsibility for making sure that the corresponding sporadic task has a higher conceptual priority than the currently executing task before preempting the currently executing task.)

Detailed Description Text (343):

c. If the most recently dispatched task blocks on an I/O request, the watchdog task (described below) will send a wakeup signal to the dispatcher. When a dispatched task blocks, the dispatcher sets its status to suspended (not ready to run). Later, when the dispatcher next decides to give this task a chance to run, it sets the status to ready to run and dispatches it. If the I/O request is still blocked, the watchdog will once again send a wakeup signal to the dispatcher and the dispatcher will once again set this application task's status to suspended.

Detailed Description Text (346):

the user thread that is scheduled for execution blocks. So the watchdog's sole responsibility is to notify the dispatcher that the application thread has gone to sleep. In response, the dispatcher will schedule another thread for execution.

Detailed Description Text (349):

2. The dispatcher task is written partially in PERC. However, it is very important that the portion of the dispatcher that responds to asynchronous "interrupts" from the watchdog task and the alarm timer be written in C. The dispatcher can only use its PERC stacks during times when it is sure that the most recently dispatched PERC task is blocked and/or suspended.

Detailed Description Text (351):

The Java implementation of the TaskDispatcher class is illustrated in FIG. 53. This class is represented by a combination of Java and native methods. The native methods provide an interface to services provided by the underlying operating system. Note that TaskDispatcher extends Thread.

Detailed Description Text (358):

5. Sets the watchdog thread's priority to THREAD.sub.-- PRIORITY.sub.-- LOWEST. The purpose of the watchdog thread is to determine when the most recently scheduled Java thread has gone to sleep or been blocked. When this happens, the watchdog thread will begin to run and it will notify the dispatcher that the most recently dispatched Java thread is no longer running.

Detailed Description Text (364):

2. startDispatcher() returns as a Java integer a Boolean flag which indicates whether garbage collection is enabled. In normal operation, garbage collection is always enabled. However, the system supports an option of disabling garbage collection so as to facilitate certain kinds of debugging and performance monitoring analyses.

Detailed Description Text (369):

Note that implementation of task priorities is provided by the nrt.sub.-- ready.sub.-- q object. Its getNextThread() method always returns the highest priority thread that is ready to run. Note also that it would be straightforward to modify this code so that the duration of each thread's time slice is variable. Some thread's might require CPU time slices that are longer than 25 ms and others might tolerate time slices that are shorter. runThread() (See FIG. 54) performs the following:

Detailed Description Text (371):

2. Saves the Java state of the executing dispatcher thread by executing the SaveThreadState() macro (See FIG. 44).

Detailed Description Text (376):

a. An event is triggered by the watchdog task, or by the task executing its relinquish() method (See FIG. 54). This event will be triggered if the dispatched task blocks (on I/O or sleep, for example).

Detailed Description Text (380):

b. Waiting for the thread to either block or to voluntarily preempt itself (See relinquish() in FIG. 54, PreemptTask() in FIG. 43, and exitAtomic() in FIG. 97).

Detailed Description Text (386):

12. Restores the state of the dispatcher task (and returns to the Java method that invoked the native runThread() method.

Detailed Description Text (411):

7.5.1 Blocking Function and I/O System Calls

Detailed Description Text (414):

2. A blocking I/O or mutual exclusion request requires the task to be put to sleep

Detailed Description Text (417):

In the first case, the protocol described immediately above ensures that local variables are in a consistent state at the moment the task is preempted. To handle the second case, we require that any C code in the run-time system that calls a non-fast function consider all of its fast pointers to have been invalidated by invocation of the non-fast function. Further, we require that the invocation of blocking system calls be surrounded by the PrepareBlockCall() and ResumeAfterBlockCall() macros, as shown below:

Detailed Description Text (422):

1. Set the thread's execution status to MAY.sub.-- BLOCK.

Detailed Description Text (426):

2. Checks a thread state variable to see if an asynchronous exception was sent to this thread while it was blocked. If so, the macro throws the exception.

Detailed Description Text (428):

What if the dispatcher awakes to trigger a preemption immediately after the task has blocked, but before the watchdog has notified the dispatcher that the most recently dispatched task blocked on an I/O request? In this case, the dispatcher would set the task's thread state to indicate that a preemption is requested. Then the dispatcher would wait for the task to preempt itself. If the task continues to

be blocked, the watchdog will notify the dispatcher that the task is now blocked, and the dispatcher will retract its preemption request and mark the thread as having been blocked. If, on the other hand, the task becomes unblocked after the dispatcher awakes but before the watchdog has a chance to complete its notification of the dispatcher that this task had been blocked, the watchdog will not complete its notification of the dispatcher and the

Detailed Description Text (429):

dispatcher will never know the task was ever blocked. The task will be allowed to continue execution up to its next preemption point before it is preempted.

Detailed Description Text (431):

Native libraries are implemented according to a protocol that allows references to dynamic objects to be automatically updated whenever the dynamic object is relocated by the garbage collector. However, if these native libraries call system routines which do not follow the native-library protocols, then the system routines are likely to become confused when the corresponding objects are moved. To avoid this problem, programmers who need to pass heap pointers to system libraries must make a stable copy of the heap object and pass a pointer to the stable copy. The stable copy should be allocated on the C stack, as a local variable. If necessary, upon return from the system library, the contents of the stable copy should be copied back into the heap. Note that on uniprocessor systems a non-portable performance optimization to this strategy is possible when invoking system libraries that are known not to block if thread preemption is under PERC's control. In particular, we can pass the system library a pointer to the dynamic object and be assured that the dynamic object will not be relocated (since the garbage collector will not be allowed to run) during execution of the system library routine.

Detailed Description Text (495):

Every PERC object begins with two special fields representing the object's lock and method tables respectively. See FIG. 23 for the declaration of MethodTable. The method table's first field is a pointer to the corresponding Class object. The second field is a pointer to an array of pointers to Method objects. The third field is a pointer to the JIT-code implementation of the first method, followed by a pointer to the JIT-code implementation of the second method, and so on. The pointers to JIT-code implementations may actually be pointers only to stub procedures that interface JIT code to byte-code or native-code methods.

Detailed Description Text (499):

Note that every allocated object is tagged according to which real-time activity allocated it. This is necessary in order to allow the run-time system to enforce memory allocation budgets for each activity. Allocations performed by traditional Java applications that are not executing as part of a real-time activity are identified by a null-valued Activity pointer. All of the allocate routines consult the Thread referenced by `sub.-- current.sub.-- thread` to determine which Activity the current thread belongs to.

Detailed Description Text (521):

String and substring data is special in that we may have arrays of bytes that are shared by multiple overlapping strings. The bytes themselves are represented in a block of memory known to the garbage collector as a String. The programmer represents each string using a String object. FIG. 7 shows string objects `x` and `y`, representing the strings "embedded" and "bed" respectively. The value field of each string object is a pointer to the actual string data. The offset field is the offset, measured in bytes, of the start of the string within the corresponding StringData buffer. The count field is the number of bytes in the string. Note that count represents bytes, even though Unicode strings might require two bytes to represent each character.

Detailed Description Text (553):

In Java, programmers can specify an action to be performed when objects of certain types are reclaimed by the garbage collector. These actions are specified by including a non-empty finalize method in the class definition. Such objects are said to be finalizable. When a finalizable object is allocated, the two low order bits of the Activity Pointer are set to indicate that the object is finalizable. The 0.times.01 bit, known symbolically as FINAL.sub.-- LINK, signifies that this object has an extra Finalize Link field appended to the end of it. The 0.times.02 bit, known symbolically as FINAL.sub.-- OBJ, signifies that this object needs to be finalized. After the object has been finalized once, its FINAL.sub.-- OBJ is cleared, but its FINAL.sub.-- LINK bit remains on throughout the object's lifetime.

Detailed Description Text (608):

The standard model for execution of Java byte-code programs assumes an execution model comprised of a single stack. Furthermore, the Java byte codes are designed to support dynamic loading and linking. This requires the use of symbolic references to external symbols. Resolving these symbolic references is a fairly costly operation which should not be performed each time an external reference is accessed. Instead, the PERC virtual machine replaces symbolic references with more efficient integer index and direct pointer references when the code is loaded.

Detailed Description Text (609):

In order to achieve good performance, the PERC virtual machine does not check for type correctness of arguments each time it executes a byte-code instruction. Rather, it assumes that the supplied arguments are of the appropriate type. Since byte-code programs may be downloaded from remote computer systems, some of which are not necessarily trustworthy, it is necessary for the PERC virtual machine to scrutinize the byte-code program for type correctness before it begins to execute. The process of guaranteeing that all of the operands supplied to each byte-code instruction are of the appropriate type is known as byte code verification. Once the types of each operation are known, it is possible to perform certain code transformations. Some of these transformations are designed simply to improve performance. In other cases, the transformations are needed to comply with the special requirements of the PERC virtual machine's stack protocols. For example, Java's dup2 byte code duplicates the top two elements on the Java stack. Byte-code verification determines the types of the top two stack elements. If both are of type pointer, the class loader replaces this byte code with a special instruction named dup2.sub.-- 11, which duplicates the top two elements of the pointer stack. If the two stack arguments are both non-pointer values, the PERC class loader replaces this byte code with the dup2.sub.-- 00 instruction, which duplicates the top two elements of the non-pointer stack. If one of dup's stack arguments is a pointer and the other is a non-pointer (in either order), the PERC class loader replaces dup with dup2.sub.-- 10, which duplicates the top element on each stack. A complete list of all the transformations that are performed by the byte code loader is provided in the remainder of this section.

Detailed Description Text (611):

Byte code verification is performed in two passes. In the first pass, we divide the program into basic blocks and build a data structure that identifies how control flows between basic blocks. A basic block is a straight-line sequence of instructions that does not include any branches into or out. The result of performing this first pass is known as a control-flow graph. The process of creating the control-flow graph is straightforward, and has been described in numerous references. See, for example, "Compilers: Principles, Techniques, and Tools", written by Alfred V. Aho, Ravi Sethi, and Jeffrey Ullman, published in 1988.

Detailed Description Text (612):

During construction of the control-flow-graph, we give special attention to the

basic blocks that are targeted by jsr and jsr.sub.-- w instructions. These blocks represent the starting points for the bodies of finally statements and they receive special treatment.

Detailed Description Text (613):

Before starting the second pass, we identify each of the entry points to the method. We consider the first basic block in the method to be the main entry point. Additionally, we consider the starting block for each finally statement to represent an entry point. And further, we consider the starting block for each exception handler to represent an entry point. Exception handlers are identified in the method's code attribute, in the field named exception.sub.-- table. The relevant data structures are described in "The Java Virtual Machine Specification", written by Tim Lindholm and Frank Yellin, published in 1996.

Detailed Description Text (614):

Each basic block is represented by a data structure with fields representing the following information:

Detailed Description Text (615):

1. The offsets within the method's byte code of the instructions that represent the start and end of the basic block.

Detailed Description Text (616):

2. A list of pointers to the basic block objects that may branch to this block. We call these blocks the predecessors.

Detailed Description Text (617):

3. A list of pointers to the basic block objects that this block may branch to. We call these blocks the successors.

Detailed Description Text (618):

4. A flag that signals whether this basic block has been processed by the second pass.

Detailed Description Text (619):

5. A representation of the types of the values that will be present on the Java stack at the start of executing this block.

Detailed Description Text (620):

6. A representation of the types of the values that will be present on the Java stack at the end of executing this block.

Detailed Description Text (621):

7. An integer that identifies which entry point reaches this basic block. If a particular basic block is reached by multiple entry points, the byte-code program is considered to be invalid.

Detailed Description Text (622):

The second pass consists of examining each entry point and all of the blocks reachable from that entry point, calculating the effects that each block has on the run-time stack and verifying that the proper types sit on the stack for each byte-code instruction that is executed.

Detailed Description Text (623):

Consider analysis of the main entry point and the blocks reachable from this entry point. First, we initialize the entry point's initial stack to empty. Then we simulate execution of the entry block's instructions and record the effects of these instructions in terms of the types of the values that will be popped from and pushed onto the stack. After simulating all of the instructions in this basic block, we examine each of the entry block's successors as follows:

Detailed Description Text (624):

1. If the successor has already been analyzed, we simply verify that it is identified as having been reached from this same entry point and that its notion of initial stack types is the same as this block's notion of ending stack types.

Detailed Description Text (625):

2. Otherwise, we mark the successor as analyzed, identifying it as having been reached from the same entry point that reached this block, initialize its initial stack types to be the same as this block's ending stack types, and recursively analyze this successor node using the same technique that was used to analyze the entry point.

Detailed Description Text (626):

The process continues as outlined above until all of the blocks reachable from the initial entry point have been analyzed.

Detailed Description Text (630):

Most of the operations that access the constant pool can be replaced with fast variants. When a Java class is loaded into the Java virtual machine, all of the constants associated with each method are loaded into a data structure known as the constant pool. Because Java programs are linked together at run time, many constants are represented symbolically in the byte code. Once the program has been loaded, the symbolic values are replaced in the constant pool with the actual constants they represent. We call this process "resolving constants." Sun Microsystems Inc.'s descriptions of their Java implementation suggest that constants should be resolved on the fly: each constant is resolved the first time it is accessed by user code. Sun Microsystems Inc.'s documents further suggest that once an instruction making reference to a constant value has been executed and the corresponding constant has been resolved, that byte code instruction should be replaced with a quick variant of the same instruction. The main difference between the quick variant and the original instruction is that the quick variant knows that the corresponding constant has already been resolved.

Detailed Description Text (673):

The standard Java byte code assumes that all local variables and all push and pop operations refer to a single shared stack. Offsets for local variables are all calculated based on this assumption. Our implementation maintains two stacks, one for non-pointers and another for pointers. Pointer local variables are stored on the pointer stack. And non-pointer locals are stored on the non-pointer stack. Thus, our byte-code loader has to remap the offsets for all local variable operations. The affected instructions are: iload, iload.sub.-- <n>, lload, lload.sub.-- <n>, fload, fload.sub.-- <n>, dload, dload.sub.-- <n>, aload, aload.sub.-- <n>, istore, istore.sub.-- <n>, lstore, lstore.sub.-- <n>, fstore, fstore.sub.-- <n>, dstore, dstore.sub.-- <n>, astore, astore.sub.-- <n>, iinc.

Detailed Description Text (749):

1. The code used in the implementation of the ROMizer tool to read in a Java class file, verify the validity of the byte code, and transform the byte code into the PERC instruction set is the exact same code that is used by the PERC implementation to support dynamic (on-the-fly) loading of new byte-code functionality into the PERC virtual machine.

Other Reference Publication (1):

Singhal et al., Building high performance applications and services in Java: An experimental study, ACM, pp. 16-20, 1997.

Other Reference Publication (2):

Buss et al., "Discrete event simulation on the world wide web using Java", Proc of 1996 winter simulation conf., ACM, pp. 780-785, 1996.

Other Reference Publication (3):

Sundaresen et al., "Java paradigms for mobile agent facilities", OOPSLA ACM, pp. 133-135, 1997.

Other Reference Publication (7):

Nair et al, "Java based query driven simulation environment", Proc. of 1996 winter simulation conf., pp. 786-793, 1996.

Other Reference Publication (8):

Don Brutzman, "The virtual reality modeling language and Java", Comm. of the ACM, vol. 41, No. 6, pp. 57-64, Jun. 1998.

CLAIMS:

1. A real-time virtual machine method (RTVMM) for implementing real-time systems and activities, the RTVMM comprising the steps:

implementing an O-OPL program that can run on computer systems of different designs, an O-OPL program being based on an object-oriented programming language (O-OPL) comprising object type declarations called classes, each class definition describing the variables that are associated with each object of the corresponding class and all of the operations called methods that can be applied to instantiated objects of the specified type, a "method" being a term of art describing the unit of procedural abstraction in an object-oriented programming system, an O-OPL program comprising one or more threads wherein the run-time stack for each thread is organized so as to allow accurate identification of type-tagged pointers contained on the stack without requiring type tag information to be updated each time the stack's content changes, the O-OPL being an extension of a high-level language (HLL) exemplified by Java, HLL being an extension of a low-level language (LLL) exemplified by C and C++, a thread being a term of art for an independently-executing task, an O-OPL program being represented at run time by either O-OPL byte codes or by native machine codes.

59. The RTVMM of claim 1 wherein each object has a "lock" field initialized to a "null" value, the implementing step comprising the steps:

causing a "hashlock object" to be allocated memory space if the "lock" field of the object contains a "null" value;

causing the next available hash value to be identified;

causing the "hash value" field of the "hashlock object" to be initialized to the next available hash value;

causing the "lock" field of the object to be initialized to refer to the newly-allocated "hashlock object".

60. The RTVMM of claim 59 wherein the implementing step comprises the step:

causing the "hash value" field of the "hashlock object" to be overwritten to the next available hash value if the "lock" field of the object does not have a "null" value and if the "hash value" field has the value zero.

62. The RTVMM of claim 1 wherein each object has a "lock" field initialized to a "null" value, the implementing step comprising the steps:

causing a "hashlock object" for each object needing either a lock or a hash value to be allocated memory space and initialized, the "hashlock object" having a "hash value" field;

causing the address of the "hashlock object" to be written into the "lock" field of the object;

causing the hash value of an object to be retrieved by reading the "hash value" field of the associated "hashlock object".

63. The RTVMM of claim 62 wherein a monitor object is to be accessed and a "lock" field of the monitor object has a "null" value, the implementing step comprising the steps:

causing memory space to be allocated for a "hashlock object";

causing a "count" field of the "hashlock object" to be initialized to 1;

causing a "u-owner" field of the "hashlock object" to be set to represent the current thread;

causing access to be granted to the monitor object.

64. The RTVMM of claim 62 wherein a monitor object is to be accessed and a "lock" field of the monitor does not have a "null" value thereby indicating the existence of a "hashlock object", the implementing step comprising the steps:

causing a "count" field of the "hashlock object" to be incremented;

causing a "u-owner" field of the "hashlock object" to be set to represent the current thread;

causing access to be granted to the monitor object; provided the "count" field is 0 or the "u-owner" field refers to the currently-executing thread; otherwise:

causing the currently-executing thread to be placed on a waiting list queue;

causing the execution of the currently-executing thread to be blocked until access can be granted to the monitor object.

65. The RTVMM of claim 62 wherein threads are assigned priorities and a higher-priority thread's access to an object is being blocked by a lower-priority thread, the implementing step comprising the step:

causing the priority of the higher-priority thread to be assigned to the lower-priority thread until the lower-priority thread releases its lock on the object.

66. The RTVMM of claim 62 wherein a thread requests access to a monitor object be terminated, the implementing step comprising the steps:

causing verification that a "u-owner" field of the "hashlock" object associated with the monitor object represents the thread;

causing a "count" field in the "hashlock" object to be decremented; if the new value in the "count" field is zero, then:

causing the "u-owner" field of the "hashlock" object to be set to represent the highest-priority member of a waiting list for the monitor object;

causing the "count" field of the "hashlock" object to be set to 1;

causing the removal of the highest-priority member of the waiting list for the monitor object;

provided the waiting list is not empty; otherwise:

causing the "u-owner" field of the "hashlock" object to be set to a "null" value.

67. The RTVMM of claim 62 wherein a thread's access to a monitor object has been terminated and a "hash value" field of a "hashlock object" associated with the monitor object is 0, the implementing step comprising the steps:

causing a "lock" field in the monitor object to be set to a "null" value;

causing the placement of the "hashlock object" on a list of available "hashlock objects" to be used in satisfying new requests for "hashlock objects".

69. The RTVMM of claim 1 wherein the implementing step comprises the steps:

creating a normally-sleeping thread called a thread dispatcher;

causing the thread dispatcher to be awakened if an interrupt arrives from an alarm timer that has determined that a specified time period has expired, the thread dispatcher then suspending execution of the currently-executing thread;

causing the thread dispatcher to be awakened if an interrupt arrives indicating the necessity of preempting the currently-executing thread so that a sporadic task can be executed;

causing the thread dispatcher to be awakened if the currently-executing thread blocks on an I/O request, the thread dispatcher then suspending execution of the currently-executing thread.

70. The RTVMM of claim 69 wherein the implementing step comprises the step:

creating a watchdog thread that sends an interrupt to the thread dispatcher when a thread that is scheduled for execution blocks.

83. The RTVMM of claim 1 wherein the implementing step comprises the steps:

creating a thread called a thread dispatcher;

creating a watchdog thread that sends an interrupt to the thread dispatcher when a thread that is scheduled for execution blocks, the thread dispatcher then scheduling another thread for execution.

Refine Search

Search Results -

Term	Documents
(1 AND 2).USPT.	1
(L1 AND L2).USPT.	1

Database:

US Pre-Grant Publication Full-Text Database
 US Patents Full-Text Database
 US OCR Full-Text Database
 EPO Abstracts Database
 JPO Abstracts Database
 Derwent World Patents Index
 IBM Technical Disclosure Bulletins

Search:

L3

Refine Search

Recall Text

Clear

Interrupt

Search History

 DATE: Thursday, May 20, 2004 [Printable Copy](#) [Create Case](#)

<u>Set</u> <u>Name</u> side by side	<u>Query</u>	<u>Hit</u> <u>Count</u>	<u>Set</u> <u>Name</u> result set
<i>DB=USPT; PLUR=YES; OP=ADJ</i>			
<u>L3</u>	l1 and L2	1	<u>L3</u>
<u>L2</u>	terminat\$	589530	<u>L2</u>
<u>L1</u>	JAVA and thread near2 stopped and exception and unlock near2 monitor\$1	1	<u>L1</u>

END OF SEARCH HISTORY

09/046, 069

First Hit Fwd Refs**End of Result Set**☐ **Generate Collection** **Print**

L14: Entry 1 of 1

File: USPT

Mar 23, 2004

DOCUMENT-IDENTIFIER: US 6711739 B1

TITLE: System and method for handling threads of execution

Brief Summary Text (8):

Two of these methods are inherently problematic. It may be unsafe to use the stop () method because, when a thread is terminated, objects which may have been locked by the thread are all unlocked, regardless of whether or not they were in consistent states. These inconsistencies can lead to arbitrary behavior which make corrupt the functions of the thread and the program. The suspend() method may also cause problems because it is prone to deadlock. For example, if a thread holds a lock on the monitor for a particular resource when it is suspended, no other thread can access the resource until the thread holding the lock is resumed. If a second thread should cause the first thread to resume, but first attempts to access the resource, it may wait for access to the resource and consequently may never call the first thread's resume() method. Because these two methods are so problematic, their use is discouraged and they are being deprecated from the Java Developers Kit produced by Java's originator, Sun Microsystems Inc.

Detailed Description Text (24):

Referring to FIG. 2, a flow diagram illustrating the execution of instructions in a thread is shown. In this example, the thread is configured to execute instructions 1-N, then loop back and repeat these instructions. The thread will continue to execute instructions 1-N until the thread is stopped. If the Thread.stop() method is invoked to stop the thread, it will immediately cause an exception and terminate the thread's execution. The stop() method is not constrained to halt execution of the thread at any particular point, so there is no way to determine where among these instructions execution will be stopped. The thread may be stopped at point A, point B, point C, or any other point in the execution of the thread's run() method. If the instructions modify the state of the application, stopping the thread may leave the application in an unknown state. The thread will also unlock all of the monitors which had been locked by the thread, possibly leaving objects protected by these monitors in inconsistent states. Objects which are left in these inconsistent states are said to be damaged, and operations on these damaged objects may lead to arbitrary behavior. Errors that are caused by the behavior of damaged objects may be difficult to detect and a user may have no warning that the errors have occurred.

Detailed Description Text (50):

The stop() method of the Handler class provides a means for gracefully terminating the thread and should not be overridden by the second class. By providing an indication that the thread should be stopped rather than simply stopping the thread using the stop() method of the thread class, the instability inherent in the Thread.stop() method is avoided. The thread does not immediately throw an exception (unlocking monitors as the exception propagates up the stack,) but instead allows the thread to terminate normally, leaving the system in a stable state.

[First Hit](#) [Fwd Refs](#)**End of Result Set**

Generate Collection

Print

L16: Entry 1 of 1

File: USPT

Mar 23, 2004

DOCUMENT-IDENTIFIER: US 6711739 B1

TITLE: System and method for handling threads of execution

Brief Summary Text (8):

Two of these methods are inherently problematic. It may be unsafe to use the stop () method because, when a thread is terminated, objects which may have been locked by the thread are all unlocked, regardless of whether or not they were in consistent states. These inconsistencies can lead to arbitrary behavior which make corrupt the functions of the thread and the program. The suspend() method may also cause problems because it is prone to deadlock. For example, if a thread holds a lock on the monitor for a particular resource when it is suspended, no other thread can access the resource until the thread holding the lock is resumed. If a second thread should cause the first thread to resume, but first attempts to access the resource, it may wait for access to the resource and consequently may never call the first thread's resume() method. Because these two methods are so problematic, their use is discouraged and they are being deprecated from the Java Developers Kit produced by Java's originator, Sun Microsystems Inc.

Brief Summary Text (10):

One or more of the problems outlined above may be solved by various embodiments of the present mechanism. The mechanism provides a means for controlling threads in a Java application while avoiding the unsafe conditions inherent in the use of existing java.lang.Thread methods. The present mechanism provides a simple and easy-to-use mechanism. for stopping threads without causing unnecessary waiting, without creating a need for exception handling, and without leaving the associated application in an unknown state.

Brief Summary Text (12):

When an object is instantiated from the subclass, the start() method inherited from the class is configured to create a thread having the object as its target. The start() method is also configured to set the target variable (which is local to the thread) is set to a value which indicates that the thread should be running. The stop() method of the class is also inherited by the subclass. When the stop() method is invoked, it is configured to set the target variable to a value which indicates that the thread should be stopped. The run() method provided by the subclass periodically checks the target variable within the thread. The checking of the target variable occurs in the normal course of execution of the run method. If the target variable indicates that the thread should be stopped, the run() method is configured to complete execution and exit normally, causing the thread to terminate. An exception is not required to stop the run() method, so exception handling is not necessary.

Detailed Description Text (12):

The example1 subclass extends the Thread class and inherits all of Thread's methods, except that example1 implements a run() method that overrides the run() method of the Thread class. The example1.start() method causes the Java virtual machine to execute example1.run() in a new thread of execution. In the second instance, a class implements Runnable:

Detailed Description Text (24):

Referring to FIG. 2, a flow diagram illustrating the execution of instructions in a thread is shown. In this example, the thread is configured to execute instructions 1-N, then loop back and repeat these instructions. The thread will continue to execute instructions 1-N until the thread is stopped. If the Thread.stop() method is invoked to stop the thread, it will immediately cause an exception and terminate the thread's execution. The stop() method is not constrained to halt execution of the thread at any particular point, so there is no way to determine where among these instructions execution will be stopped. The thread may be stopped at point A, point B, point C, or any other point in the execution of the thread's run() method. If the instructions modify the state of the application, stopping the thread may leave the application in an unknown state. The thread will also unlock all of the monitors which had been locked by the thread, possibly leaving objects protected by these monitors in inconsistent states. Objects which are left in these inconsistent states are said to be damaged, and operations on these damaged objects may lead to arbitrary behavior. Errors that are caused by the behavior of damaged objects may be difficult to detect and a user may have no warning that the errors have occurred.

Detailed Description Text (26):

Referring to FIG. 3, a flow diagram illustrating the execution of instructions in a thread using the present mechanism is shown. After the start() method is called, the body of the run() method is executed. The flow diagram on the left side of the figure represents body of the run() method. The thread is still configured to execute instructions 1-N, but it is further configured to periodically examine the target variable to determine its value (e.g., by using the isRunning() method described below.) If the target variable is set to indicate that the thread should continue running, instructions 1-N are repeated. If the target variable is set to indicate that the thread should be stopped (e.g., using the stop() method illustrated here as a different thread of execution,) the thread branches to point A, where it completes execution and exits normally. Because instructions 1-N are completed normally before the target variable is checked, the state of the application is easier to determine. Because the run() method executes to completion, no exception handling is required and no objects are damaged by abnormal termination of the thread. It should be noted that the target variable can be checked at different points in the code of the run() method. It should also be noted that the target variable can be set by other threads or by the run() method itself to indicate that the thread should be stopped.

Detailed Description Text (50):

The stop() method of the Handler class provides a means for gracefully terminating the thread and should not be overridden by the second class. By providing an indication that the thread should be stopped rather than simply stopping the thread using the stop() method of the thread class, the instability inherent in the Thread.stop() method is avoided. The thread does not immediately throw an exception (unlocking monitors as the exception propagates up the stack,) but instead allows the thread to terminate normally, leaving the system in a stable state.

Detailed Description Text (53):

The Handler class described above thereby provides the following advantages: first, it gracefully handles stopping a thread without exception handling; second, classes which extend the Handler class implement the Runnable interface and can be referenced as Runnable; third, it can be determined locally within a thread whether the thread should continue to run, or should be terminated, based on the result of the isRunning() method; and fourth, because this class has such a simple API, it provides a suitable core for Java servers, agents and socket handlers to handle asynchronous peer communications. The Handler class thereby provides a means for standardization in the handling of threads which is object oriented, which uses

self-contained logic, and which allows developers to use programming techniques with which they are already familiar. (It should be noted that other embodiments may vary from the implementation described above and may therefore provide advantages which differ from those listed here.)

CLAIMS:

1. A system, comprising: a dedicated Java thread handler class which includes: a start method that sets a target variable to indicate that a thread extending said dedicated Java thread handler class is running; an abstract run method; and a stop method that sets said target variable to indicate that said thread extending said Java class is to be stopped; wherein said dedicated Java thread handler class is configured to be extended by custom thread subclasses that each includes a run method that overrides said abstract run method of said Java class, wherein said dedicated Java thread handler class does not include functionality particular to any of the custom thread subclasses, wherein said run method of one of the custom thread subclasses: provides one or more instructions to be executed within said thread, wherein said instructions are configured to access one or more objects during execution of said thread; provides one or more checks during execution of said thread to determine if said target variable indicates that said thread is to be stopped; and completes execution of said run method if one of the one or more checks determines that said thread is to be stopped; wherein completing execution of said run method if one of the one or more checks determines that said thread is to be stopped stops the thread without throwing an exception and leaves each of said one or more objects in a consistent state.

9. A method, comprising: defining a dedicated Java thread handler class wherein said dedicated Java thread handler class defines a first method that initializes a target variable to indicate that a thread extending said dedicated Java thread handler class is running, wherein said dedicated Java thread handler class defines a second method that sets said target variable to indicate that said thread extending said dedicated Java thread handler class is to be stopped; instantiating an object that inherits said methods from said dedicated Java thread handler class, wherein said object includes a run method implementing one or more instructions to perform a particular task, wherein said dedicated Java thread handler class does not include any instructions for said particular task; creating a custom thread in which said one or more instructions are executed; starting said custom thread; calling said first method to initialize said target variable; initiating execution of said run method in said custom thread; calling said second method during execution of said run method to indicate that said custom thread is to be stopped; said run method detecting that said target variable is set to indicate that said custom thread is to be stopped; and completing execution of said run method after said detecting; wherein said completing execution of said run method stops the custom thread without throwing an exception and leaves one or more objects accessed by the custom thread in a consistent state.

13. A computer-readable storage medium containing a plurality of program instructions, wherein said program instructions define a method comprising: defining a dedicated Java thread handler class wherein said dedicated Java thread handler class defines a first method that initializes a target variable to indicate that a thread extending said dedicated Java thread handler class is running, wherein said dedicated Java thread handler class defines a second method that sets said target variable to indicate that said thread extending said dedicated Java thread handler class is to be stopped; instantiating an object that inherits said methods from said dedicated Java thread handler class, wherein said object includes a run method implementing one or more instructions to perform a particular task, wherein said dedicated Java thread handler class does not include any instructions for said particular task; creating a custom thread in which said one or more instructions are executed; starting said custom thread; calling said first method to initialize said target variable; initiating execution of said run method

in said custom thread; calling said second method during execution of said run method to indicate that said custom thread is to be stopped; said run method detecting that said target variable is set to indicate that said custom thread is to be stopped; and completing execution of said run method after said detecting; wherein said completing execution of said run method stops the custom thread without throwing an exception and leaves one or more objects accessed by the custom thread in a consistent state.

18. The system of claim 17, wherein, to terminate the thread normally, no exceptions are thrown and one or more objects accessed by said thread are left in a consistent state.

23. A method, comprising: starting execution of a custom thread which implements a subclass of a dedicated Java thread handler class, wherein said dedicated Java thread handler class includes a start method that initializes a target variable to indicate that said custom thread is running and a stop method that sets said target variable to indicate that said custom thread is to be stopped, and wherein said subclass includes a run method that includes code executed by said custom thread, wherein the thread handler class does not include functionality particular to the custom thread; calling said stop method during execution of said custom thread to indicate that said custom thread is to be stopped; said run method detecting that said target variable is set to indicate that said custom thread is to be stopped; and completing execution of said run method after said detecting; wherein said completing execution of said run method after said detecting stops the custom thread without throwing an exception and leaves objects accessed by the custom thread in a consistent state.